(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
8 March 2001 (08.03.2001)

PCT

(10) International Publication Number
WO 01/16715 A1

(51) International Patent Classification[7]: G06F 9/32, 9/26

(21) International Application Number: PCT/US00/23983

(22) International Filing Date: 31 August 2000 (31.08.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/151,961 1 September 1999 (01.09.1999) US

(63) Related by continuation (CON) or continuation-in-part (CIP) to earlier application:
US 60/151,961 (CIP)
Filed on 1 September 1999 (01.09.1999)

(71) Applicant *(for all designated States except US)*: INTEL CORPORATION [US/US]; 2200 Mission College Boulevard, Santa Clara, CA 95052 (US).

(72) Inventors; and
(75) Inventors/Applicants *(for US only)*: WOLRICH, Gilbert [US/US]; 4 Cider Mill Road, Framingham, MA 01701 (US). ADILETTA, Matthew, J. [US/US]; 20 Monticello Drive, Worcester, MA 01603 (US). WHEELER, William [US/US]; 745 School Street, Webster, MA 01570 (US). BERNSTEIN, Debra [US/US]; 38 Helen Street, Waltham, MA 02452 (US). HOOPER, Donald [US/US]; 19 Main Circle, Shrewsbury, MA 01545 (US).

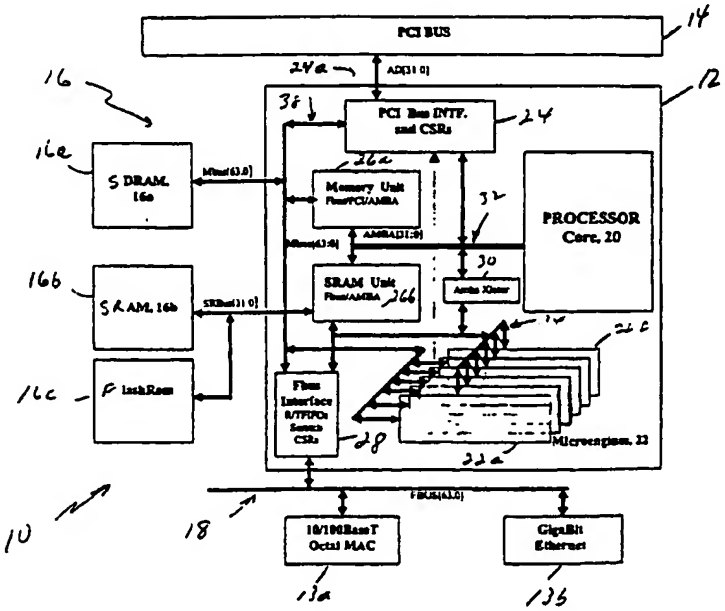
(74) Agents: MALONEY, Denis, G.; Fish & Richardson P.C., 225 Franklin Street, Boston, MA 02110-2804 et al. (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CR, CU, CZ, DE, DK, DM, DZ, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

[Continued on next page]

(54) Title: BRANCH INSTRUCTIONS IN A MULTITHREADED PARALLEL PROCESSING SYSTEM



(57) Abstract: A processor such as a parallel hardware-based multithreaded processor (12) is described. The processor can execute a computer instruction that is a branch instruction that causes an execution stream to branch to an instruction at an address specified in the instruction if a state of a specified state name of the processor is a specified value.

WO 01/16715 A1

Published:
— *With international search report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

BRANCH INSTRUCTIONS IN A MULTITHREADED PARALLEL PROCESSING SYSTEM

BACKGROUND

This invention relates to branch instructions.

Parallel processing is an efficient form of information processing of concurrent events in a computing process. Parallel processing demands concurrent execution of many programs in a computer. Sequential processing or serial processing has all tasks performed sequentially at a single station whereas, pipelined processing has tasks performed at specialized stations. Computer code whether executed in parallel processing, pipelined or sequential processing machines involves branches in which an instruction stream may execute in a sequence and branch from the sequence to a different sequence of instructions.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 is a block diagram of a communication system employing a processor.

FIG. 2 is a detailed block diagram of the processor.

FIG. 3 is a block diagram of a microengine used in the processor of FIGS. 1 and 2.

FIG. 4 is a diagram of a pipeline in the microengine.

FIG. 5 shows exemplary formats for branch instructions.

FIG. 6 is a block diagram of general purpose registers.

DESCRIPTION

Referring to FIG. 1, a communication system 10 includes a processor 12. In one embodiment, the processor is a hardware-based multithreaded processor 12. The processor 12 is coupled to a bus such as a PCI bus 14, a memory system 16 and a second bus 18. The system 10 is especially useful for tasks that can be broken into parallel sub-tasks or functions. Specifically hardware-based multithreaded processor 12 is useful for tasks that are bandwidth oriented rather than latency oriented. The hardware-based multithreaded processor 12 has multiple microengines 22 each with multiple hardware controlled threads that can be simultaneously active and independently work on a task.

The hardware-based multithreaded processor 12 also includes a central controller 20 that assists in loading microcode control for other resources of the hardware-based multithreaded processor 12 and performs other general purpose computer type functions such as handling protocols, exceptions, extra support for packet processing where the microengines pass the packets off for more detailed processing such as in boundary conditions. In one embodiment, the processor 20 is a Strong Arm® (Arm is a trademark of ARM Limited, United Kingdom) based architecture. The general purpose microprocessor 20 has an operating system. Through the operating system the processor 20 can call functions to operate on microengines 22a-22f. The processor 20 can use any supported operating system preferably a real time operating system. For the core processor implemented as a Strong Arm architecture, operating systems such as, MicrosoftNT® real-time, VXWorks and □CUS, a freeware operating system available over the Internet, can be used.

The hardware-based multithreaded processor 12 also includes a plurality of function microengines 22a-22f. Functional microengines (microengines) 22a-22f each maintain a plurality of program counters in hardware and states associated with the program counters. Effectively, a corresponding plurality of sets of threads can be simultaneously active on each of the microengines 22a-22f while only one is actually operating at any one time.

Microengines 22a-22f each have capabilities for processing four hardware threads. The microengines 22a-22f operate with shared resources including memory system 16 and bus interfaces 24 and 28. The memory system 16 includes a Synchronous Dynamic Random Access Memory (SDRAM) controller 26a and a Static Random Access Memory (SRAM) controller 26b. SDRAM memory 16a and SDRAM controller 26a are typically used for processing large volumes of data, e.g., processing of network payloads from network packets. The SRAM controller 26b and SRAM memory 16b are used in, e.g., networking packet processing, postscript processor, or as a processor for a storage subsystem, i.e., RAID disk storage, or for low latency, fast access tasks, e.g., accessing look-up tables, memory for the core processor 20, and so forth.

The processor 12 includes a bus interface 28 that couples the processor to the second bus 18. Bus interface 28 in one embodiment couples the processor 12 to the so-called FBUS 18 (FIFO bus). The processor 12 includes a second interface e.g., a PCI bus interface 24 that couples other system components that reside on the PCI 14 bus to the processor 12. The PCI bus interface 24, provides a high speed data path 24a to the SDRAM memory 16a. Through that path data can be moved quickly from the SDRAM 16a through the PCI bus 14, via direct memory access (DMA) transfers.

Each of the functional units are coupled to one or more internal buses. The internal buses are dual, 32 bit buses (i.e., one bus for read and one for write). The hardware-based multithreaded processor 12 also is constructed such that the sum of the bandwidths of the internal buses in the processor 12 exceed the bandwidth of external buses coupled to the processor 12. The processor 12 includes an internal core processor bus 32, e.g., an ASB bus (Advanced System Bus) that couples the processor core 20 to the memory controller 26a, 26c and to an ASB translator 30 described below. The ASB bus is a subset of the so called AMBA bus that is used with the Strong Arm processor core. The processor 12 also includes a private bus 34 that couples the microengine units to SRAM controller 26b, ASB translator 30 and FBUS interface 28. A memory bus 38 couples the memory controller 26a, 26b to the bus interfaces 24 and 28 and memory system 16 including flashrom 16c used for boot operations and so forth.

Referring to FIG. 2, each of the microengines 22a-22f includes an arbiter that examines flags to determine the available threads to be operated upon. Any thread from any of the microengines 22a-22f can access the SDRAM controller 26a, SDRAM controller 26b or FBUS interface 28. The memory controllers 26a and 26b each include a plurality of queues to store outstanding memory reference requests. The FBUS interface 28 supports Transmit and Receive flags for each port that a MAC device supports, along with an Interrupt flag indicating when service is warranted. The FBUS interface 28 also includes a controller 28a that performs header processing of incoming packets from the FBUS 18. The controller 28a extracts the packet headers and performs a microprogrammable source/destination/protocol hashed lookup (used for address smoothing) in SRAM.

The core processor 20 accesses the shared resources. The core processor 20 has a direct communication to the SDRAM controller 26a to the bus interface 24 and to SRAM controller 26b via bus 32. However, to access the microengines 22a-22f and transfer registers located at any of the microengines 22a-22f, the core processor 20 access the microengines 22a-22f via the ASB Translator 30 over bus 34. The ASB translator 30 can physically reside in the FBUS interface 28, but logically is distinct. The ASB Translator 30 performs an address translation between FBUS microengine transfer register locations and core processor addresses (i.e., ASB bus) so that the core processor 20 can access registers belonging to the microengines 22a-22c.

Although microengines 22 can use the register set to exchange data as described below, a scratchpad memory 27 is also provided to permit microengines to write data out to the memory for other microengines to read. The scratchpad 27 is coupled to bus 34.

The processor core 20 includes a RISC core 50 implemented in a five stage pipeline performing a single cycle shift of one operand or two operands in a single cycle, provides multiplication support and 32 bit barrel shift support. This RISC core 50 is a standard Strong Arm® architecture but it is implemented with a five stage pipeline for performance reasons. The processor core 20 also includes a 16 kilobyte instruction cache 52, an 8 kilobyte data cache 54 and a prefetch stream buffer 56. The core processor 20 performs arithmetic operations in parallel with memory writes and instruction fetches. The core processor 20 interfaces with other functional units via the ARM defined ASB bus. The ASB bus is a 32-bit bi-directional bus 32.

Referring to FIG. 3, an exemplary microengine 22f includes a control store 70 that includes a RAM which stores a microprogram. The microprogram is loadable by the core processor 20. The microengine 22f also includes controller logic 72. The controller logic includes an instruction decoder 73 and program counter (PC) units 72a-72d. The four micro program counters 72a-72d are maintained in hardware. The microengine 22f also includes context event switching logic 74. Context event logic 74 receives messages (e.g., SEQ_#_EVENT_RESPONSE; FBI_EVENT_RESPONSE; SRAM _EVENT_RESPONSE; SDRAM _EVENT_RESPONSE; and ASB _EVENT_RESPONSE) from each one of the shared resources, e.g., SRAM 26a, SDRAM 26b, or processor core 20, control and status registers, and so forth. These messages provide information on whether a requested function has completed. Based on whether or not a function requested by a thread has completed and signaled completion, the thread needs to wait for that completion signal, and if the thread is enabled to operate, then the thread is placed on an available thread list (not shown). The microengine 22f can have a maximum of e.g., 4 threads available.

In addition to event signals that are local to an executing thread, the microengines 22 employ signaling states that are global. With signaling states, an executing thread can broadcast a signal state to all microengines 22. Receive Request or Available signal, any and all threads in the microengines can branch on these signaling states. These signaling states can be used to determine availability of a resource or whether a resource is due for servicing.

The context event logic 74 has arbitration for the four (4) threads. In one embodiment, the arbitration is a round robin mechanism. Other techniques could be used including priority queuing or weighted fair queuing. The microengine 22f also includes an execution box (EBOX) data path 76 that includes an arithmetic logic unit 76a and general purpose register set 76b. The arithmetic logic unit 76a performs arithmetic and logical functions as well as shift functions. The arithmetic logic unit includes condition code bits that are used by instructions described below. The registers set 76b has a relatively large number of general purpose registers that are windowed as will be described so that they are relatively and absolutely addressable. The microengine 22f also includes a write transfer register stack 78 and a read transfer stack 80. These registers are also windowed so that they are relatively and absolutely addressable. Write transfer register stack 78 is where write data to a resource is located. Similarly, read register stack 80 is for return data from a shared resource. Subsequent to or concurrent with data arrival, an event signal from the respective shared resource e.g., the SRAM controller 26a, SDRAM controller 26b or core processor 20 will be provided to context event arbiter 74 which will then alert the thread that the data is available or has been sent. Both transfer register banks 78 and 80 are connected to the execution box (EBOX) 76 through a data path.

Referring to FIG. 4, the microengine datapath maintains a 5-stage micro-pipeline 82. This pipeline includes lookup of microinstruction words 82a, formation of the register file addresses 82b, read of operands from register file 82c, ALU, shift or compare operations 82d, and write-back of results to registers 82e. By providing a write-back data bypass into the ALU/shifter units, and by assuming the registers are implemented as a register file (rather than a RAM), the microengine can perform a simultaneous register file read and write, which completely hides the write operation.

The instruction set supported in the microengines 22a-22f support conditional branches. The worst case conditional branch latency (not including jumps) occurs when the branch decision is a result of condition codes being set by the previous microcontrol instruction. The latency is shown below in Table 1:

TABLE 1

	1	2	3	4	5	6	7	8
microstore lookup	n1	cb	n2	XX	b1	b2	b3	b4
reg addr gen		n1	cb	XX	XX	b1	b2	b3
reg file lookup			n1	cb	XX	XX	b1	b2
ALU/shifter/cc				n1	cb	XX	XX	b1
write back			m2	n1	cb	XX	XX	

where nx is pre-branch microword (n1 sets cc's), cb is conditional branch, bx is post-branch microword and XX is an aborted microword

As shown in Table 1, it is not until cycle 4 that the condition codes of n1 are set, and the branch decision can be made (which in this case causes the branch path to be looked up in cycle 5). The microengine 22f incurs a 2-cycle branch latency penalty because it must abort operations n2 and n3 (the 2 microwords directly after the branch) in the pipe, before the branch path begins to fill the pipe with operation b1. If the branch is not taken, no microwords are aborted and execution continues normally. The microengines have several mechanisms to reduce or eliminate the effective branch latency.

The microengines support selectable deferred branches. Selectable deferring branches are when a microengine allows 1 or 2 micro instructions after the branch to execute before the branch takes effect (i.e. the effect of the branch is "deferred" in time). Thus, if useful work can be found to fill the wasted cycles after the branch microword, then the branch latency can be hidden. A 1-cycle deferred branch is shown below in Table 2 where n2 is allowed to execute after cb, but before b1:

TABLE 2

	1	2	3	4	5	6	7	8
microstore lookup	n1	cb	n2	XX	b1	b2	b3	b4
reg addr gen		n1	cb	n2	XX	b1	b2	b3
reg file lookup			n1	cb	n2	XX	b1	b2
ALU/shifter/cc				n1	cb	n2	XX	b1
write back					n1	cb	n2	XX

A 2-cycle deferred branch is shown in TABLE 3 where n2 and n3 are both allowed to complete before the branch to b1 occurs. Note that a 2-cycle branch deferment is only allowed when the condition codes are set on the microword preceding the branch.

TABLE 3

	1	2	3	4	5	6	7	8	9
microstore lookup	n1	cb	n2	n3	b1	b2	b3	b4	b5
reg addr gen		n1	cb	n2	n3	b1	b2	b3	b4
reg file lkup			n1	cb	n2	n3	b1	b2	b3
ALU/shftr/cc				n1	cb	n2	n3	b1	b2
write back					n1	cb	n2	n3	b1

The microengines also support condition code evaluation. If the condition codes upon which a branch decision are made are set 2 or more microwords before the branch, then 1 cycle of branch latency can be eliminated because the branch decision can be made 1 cycle earlier as in Table 4.

TABLE 4

	1	2	3	4	5	6	7	8
microstore lookup	n1	n2	cb	XX	b1	b2	b3	b4
reg addr gen		n1	n2	cb	XX	b1	b2	b3
reg file lookup			n1	n2	cb	XX	b1	b2
ALU/shifter/cc				n1	n2	cb	XX	b1
write back					n1	n2	cb	XX

In this example, n1 sets the condition codes and n2 does not set the conditions codes. Therefore, the branch decision can be made at cycle 4 (rather than 5), to eliminate 1 cycle of branch latency. In the example in Table 5 the 1-cycle branch deferment and early setting of condition codes are combined to completely hide the branch latency. That is, the condition codes (cc's) are set 2 cycles before a 1-cycle deferred branch.

TABLE 5

	1	2	3	4	5	6	7	8
microstore lookup	n1	n2	cb	n3	b1	b2	b3	b4
reg addr gen		n1	n2	cb	n3	b1	b2	b3
reg file lookup			n1	n2	cb	n3	b1	b2
ALU/shifter/cc				n1	n2	cb	n3	b1
write back					n1	n2	cb	n3

In the case where the condition codes cannot be set early (i.e. they are set in the microword preceding the branch), the microengine supports branch guessing which attempts to reduce the 1 cycle of exposed branch latency that remains. By "guessing" the branch path or the sequential path, the microsequencer pre-fetches the guessed path 1 cycle before it definitely knows what path to execute. If it guessed correctly, 1 cycle of branch latency is eliminated as shown in Table 6.

TABLE 6

guess branch taken /branch is taken

	1	2	3	4	5	6	7	8
microstore lookup	n1	cb	n1	b1	b2	b3	b4	b5
reg addr gen		n1	cb	XX	b1	b2	b3	b4
reg file lookup			n1	cb	XX	b1	b2	b3
ALU/shifter/cc				n1	cb	XX	b1	b2
write back					n1	cb	XX	b1

If the microcode guessed a branch taken incorrectly, the microengine still only wastes 1 cycle as in TABLE 7

TABLE 7

guess branch taken /branch is NOT taken

	1	2	3	4	5	6	7	8
microstore lookup	n1	cb	n1	XX	n2	n3	n4	n5
reg addr gen		n1	cb	n1	XX	n2	n3	n4
reg file lookup			n1	cb	n1	XX	n2	n3
ALU/shifter/cc				n1	cb	n1	XX	n2
write back					n1	cb	n1	XX

However, the latency penalty is distributed differently when microcode guesses a branch is not taken. For guess branch NOT taken and the branch is NOT taken there are no wasted cycles as in Table 8.

Table 8

	1	2	3	4	5	6	7	8
microstore lookup	n1	cb	n1	n2	n3	n4	n5	n6
reg addr gen		n1	cb	n1	n2	n3	n4	n5
reg file lookup			n1	cb	n1	n2	n1	b4
ALU/shifter/cc				n1	cb	n1	n2	n3
write back					n1	cb	n1	n2

However for guess branch NOT taken /branch is taken there are 2 wasted cycles as in Table 9.

Table 9

	1	2	3	4	5	6	7	8
microstore lookup	n1	cb	n1	XX	b1	b2	b3	b4
reg addr gen		n1	cb	XX	XX	b1	b2	b3
reg file lookup			n1	cb	XX	XX	b1	b2
ALU/shifter/cc				n1	cb	XX	XX	b1
write back					n1	cb	XX	XX

The microengine can combine branch guessing with 1-cycle branch deferment to improve the result further. For guess branch taken with 1-cycle deferred branch/branch is taken is in Table 10.

Table 10

	1	2	3	4	5	6	7	8
microstore lookup	n1	cb	n2	b1	b2	b3	b4	b5
reg addr gen		n1	cb	n2	b1	b2	b3	b4
reg file lookup			n1	cb	n2	b1	b2	b3
ALU/shifter/cc				n1	cb	n2	b1	b2
write back					n1	cb	n2	b1

In the case above, the 2 cycles of branch latency are hidden by the execution of n2, and by correctly guessing the branch direction.

If microcode guesses incorrectly, 1 cycle of branch latency remains exposed as in Table 11 (guess branch taken with 1-cycle deferred branch/branch is NOT taken).

Table 11

	1	2	3	4	5	6	7	8	9
microstore lookup	n1	cb	n2	XX	n3	n4	n5	n6	n7
reg addr gen		n1	cb	n2	XX	n3	n4	n5	n6
reg file lkup			n1	cb	n2	XX	n3	n4	n5
ALU/shftr/cc				n1	cb	n2	XX	n3	n4
write back					n1	cb	n2	XX	n3

If microcode correctly guesses a branch NOT taken, then the pipeline flows sequentially in the normal unperturbed case. If microcode incorrectly guesses branch NOT taken, the microengine again exposes 1 cycle of unproductive execution as shown in Table 12.

Table 12

guess branch NOT taken/branch is taken

	1	2	3	4	5	6	7	8	9
microstore lookup	n1	cb	n2	XX	b1	b2	b3	b4	b5
reg addr gen		n1	cb	n2	XX	b1	b2	b3	b4
reg file lkup			n1	cb	n2	XX	b1	b2	b3
ALU/shftr/cc				n1	cb	n2	XX	b1	b2
write back					n1	cb	n2	XX	b1

where nx is pre-branch microword (n1 sets cc's)
cb is conditional branch
bx is post-branch microword
XX is aborted microword

In the case of a jump instruction, 3 extra cycles of latency are incurred because the branch address is not known until the end of the cycle in which the jump is in the ALU stage (Table 13).

Table 13

	1	2	3	4	5	6	7	8	9
microstore lookup	n1	jp	XX	XX	XX	j1	j2	j3	j4
reg addr gen		n1	jp	XX	XX	XX	j1	j2	j3
reg file lkup			n1	jp	XX	XX	XX	j1	j2
ALU/shftr/cc				n1	jp	XX	XX	XX	j1
write back					n1	jp	XX	XX	XX

Referring to FIG. 5, the microengines support various branch instructions such as those that branch on condition codes. In addition, the microengines also support branch instructions that branch on a processor state.

A format as shown in FIG. 5 uses br_mask field is used to specify branch. For branch mask = 15, the extended field is used to specify the various signal and state signals as listed below.

BR_INP_STATE

The BR_INP_STATE instruction branches if a state of a specified state name is set to 1. A state is set to 1 or 0 by a microengine in the processor 10 and indicates the current processing state. The state of the state name is available to all microengines. The instruction can have the following format

br_inp_state[state_name, label#], optional_token

The field label# is a symbolic label corresponding to the address of an instruction to branch to. The field "State_name" is the state name. For example, if the state name is rec_req_avail this state indicates when set that the RCV_REQ FIFO has room available for another receive request. Other states could be used. This instruction is used to integorrate a state the microengine and to perform a branch operation based on the state.

The instruction can also include an optional_token defer 1 that causes the processor to execute the instruction following this instruction before performing the branch operation. Other architectures could support additional defer optional tokens, e.g., defer 2, defer 3 and so forth. In addition the branch operation could be to branch on state name not being set, i.e., cleared.

BR_!SIGNAL

A second branch instruction BR_!SIGNAL branches if the specified signal is deasserted. If the signal is asserted, the instruction clears the signal and does not branch. The SRAM and SDRAM signals are presented to the microengine two cycles after the last Long word is written to the transfer register. The second from last Long word is written 1 cycle after the signal. All other Long words are valid when the signal is submitted. When using this instruction, the programmer should appropriately time reading of transfer registers to ensure that proper data is read

```
Example: .xfer_order $xfer0 $xfer1 $xfer2 $xfer3
sram[read, $xfer0, op1, 0, 2], sig_done
wait#:
br_!signal[sram, wait#], guess_branch
nop;delay 1 cycle before reading $xfer0
alu[gpr0,0,b,$xfer0];valid data is written to gpr0
alu[gpr1,0,b,$xfer1];valid data is written to gpr0
self#:
br[self#].
```

The extended branch field is used for branches on various context_swapping signals. EXT_BRANCH_TYPE: extended field to BRANCH_TYPE field." EXT_BRANCH_TYPE/=<20:17>,.default=<EXT_BRANCH_TYPE/SRAM>, .validity=<.GTR[.TIME1, 0]>

The signals can include the following: SRAM = 0, SDRAM = 1, FBI = 2, INTER_THREAD = 3, AUTO_PUSH = 4, START_RECEIVE = 5, SEQ_NUM1 = 6, SEQ_NUM2 = 7, PCI = 8, GET_Q_AVAIL = B, PUT_Q_AVAIL = C, REC_REQ_AVAIL = D, PUSH_PROTECT = E and PAR_ERR = F

Each microengine 22a_22f supports multi_threaded execution of four contexts. One reason for this is to allow one thread to start executing just after another thread issues a memory reference and must wait until that reference completes before doing more work. This behavior maintains efficient hardware execution of the microengines because memory latency is significant. Stated differently, if only a single thread execution was supported, the microengines would sit idle for a significant number of cycles waiting for references to return and thereby reduce overall computational throughput. Multi_threaded execution allows an microengines to hide memory latency by performing useful independent work across several threads. Two synchronization mechanisms are supplied in order to allow a thread to issue an SRAM or SDRAM reference, and then subsequently synchronize to the point in time when that reference completes.

One mechanism is Immediate Synchronization. In immediate synchronization, the microengine issues the reference and immediately swap out that context. The context will be signaled when the corresponding reference completes. Once

signaled, the context will be swapped back in for execution when a context_swap event occurs and it is its turn to run. Thus, from the point of view of a single context's instruction stream, the microword after issuing the mem reference does not get executed until the reference completes.

A second mechanism is Delayed Synchronization. In delayed synchronization, the microengine issues the reference, and then continues to execute some other useful work independent of the reference. Some time later it could become necessary to synchronize the thread's execution stream to the completion of the issued reference before further work is performed. At this point a synchronizing microword is executed that will either swap out the current thread, and swap it back in sometime later when the reference has completed, or continue executing the current thread because the reference has already completed. Delayed synchronization is implemented using two different signaling schemes:

If the memory reference is associated with a transfer register, the signal from which the thread is triggered is generated when the corresponding transfer register valid bit is set or cleared. For example, an SRAM read which deposits data into transfer register A would be signaled when the valid bit for A is set. If the memory reference is associated with the transfer FIFO or the receive FIFO, instead of a transfer register, then the signal is generated when the reference completes in the SDRAM controller 26a. Only one signal state per context is held in the microengines scheduler, thus only one outstanding signal can exist in this scheme.

There are at least two general operational paradigms from which microcontroller micro_programs could be designed. One would be that overall microcontroller compute throughput and overall memory bandwidth are optimized at the expense of single thread execution latency. This paradigm would make sense when the system has multiple microengines executing multiple threads per microengine on unrelated data packets.

A second one is that microengine execution latency should be optimized at the expense of overall microengine compute throughput and overall memory bandwidth. This paradigm could involve execution of a thread with a real_time constraint, that is, a constraint which dictates that some work must absolutely be done by some specified time. Such a constraint requires that optimization of the single thread execution be given priority over other considerations such as memory bandwidth or overall computational

throughput. A real_time thread would imply a single microengine that executes only one thread. Multiple threads would not be handled because the goal is to allow the single real_time thread to execute as soon as possible__execution of multiple threads would hinder this ability.

Referring to FIG. 6, the two register address spaces that exist are Locally accessibly registers, and Globally accessible registers accessible by all microengines. The General Purpose Registers (GPRs) are implemented as two separate banks (A bank and B bank) whose addresses are interleaved on a word-by-word basis such that A bank registers have lsb=0, and B bank registers have lsb=1. Each bank is capable of performing a simultaneous read and write to two different words within its bank.

Across banks A and B, the register set 76b is also organized into four windows $76b_0$-$76b_3$ of 32 registers that are relatively addressable per thread. Thus, thread_0 will find its register 0 at 77a (register 0), the thread_1 will find its register_0 at 77b (register 32), thread_2 will find its register_0 at 77c (register 64), and thread_3 at 77d (register 96). Relative addressing is supported so that multiple threads can use the exact same control store and locations but access different windows of register and perform different functions. The use of register window addressing and bank addressing provide the requisite read bandwidth while using only dual ported RAMS in the microengine 22f.

These windowed registers do not have to save data from context switch to context switch so that the normal push and pop of a context swap file or stack is eliminated. Context switching here has a 0 cycle overhead for changing from one context to another. Relative register addressing divides the register banks into windows across the address width of the general purpose register set. Relative addressing allows access any of the windows relative to the starting point of the window. Absolute addressing is also supported in this architecture where any one of the absolute registers may be accessed by any of the threads by providing the exact address of the register.

Addressing of general purpose registers 78 can occur in 2 modes depending on the microword format. The two modes are absolute and relative. In absolute mode, addressing of a register address is directly specified in 7-bit source field (a6-a0 or b6-b0), as shown in Table 14:

Table 14

	7	6	5	4	3	2	1	0	
A GPR:	a6	0	a5	a4	a3	a2	a1	a0	a6=0
B GPR:	b6	1	b5	b4	b3	b2	b1	b0	b6=0
SRAM/ASB:	a6	a5	a4	0	a3	a2	a1	a0	a6=1, a5=0, a4=0
SDRAM:	a6	a5	a4	0	a3	a2	a1	a0	a6=1, a5=0, a4=1

register address directly specified in 8-bit dest field (d7-d0) Table 15:

Table 15

	7	6	5	4	3	2	1	0	
A GPR:	d7	d6	d5	d4	d3	d2	d1	d0	d7=0, d6=0
B GPR:	d7	d6	d5	d4	d3	d2	d1	d0	d7=0, d6=1
SRAM/ASB:	d7	d6	d5	d4	d3	d2	d1	d0	d7=1, d6=0, d5=0
SDRAM:	d7	d6	d5	d4	d3	d2	d1	d0	d7=1, d6=0, d5=1

If <a6:a5>=1,1, <b6:b5>=1,1, or <d7:d6>=1,1 then the lower bits are interpreted as a context-relative address field (described below). When a non-relative A or B source address is specified in the A, B absolute field, only the lower half of the SRAM/ASB and SDRAM address spaces can be addressed. Effectively, reading absolute SRAM/SDRAM devices has the effective address space; however, since this restriction does not apply to the dest field, writing the SRAM/SDRAM still uses the full address space.

In relative mode, addresses a specified address is offset within context space as defined by a 5-bit source field (a4-a0 or b4-b0)Table 16:

Table 16

	7	6	5	4	3	2	1	0	
A GPR:	a4	0	context		a3	a2	a1	a0	a4=0
B GPR:	b4	1	context		b3	b2	b1	b0	b4=0
SRAM/ASB:	ab4	0	ab3	context		b2	b1	ab0	ab4=1, ab3=0
SDRAM:	ab4	0	ab3	context		b2	b1	ab0	ab4=1, ab3=1

or as defined by the 6-bit dest field (d5-d0) Table 17:

Table 17

	7	6	5	4	3	2	1	0	
A GPR:	d5	d4	context		d3	d2	d1	d0	d5=0, d4=0
B GPR:	d5	d4	context		d3	d2	d1	d0	d5=0, d4=1
SRAM/ASB:	d5	d4	d3	context		d2	d1	d0	d5=1, d4=0, d3=0
SDRAM:	d5	d4	d3	context		d2	d1	d0	d5=1, d4=0, d3=1

If <d5:d4>=1,1, then the destination address does not address a valid register, thus, no dest operand is written back.

Other embodiments are within the scope of the appended claims.

What is claimed is:

1. A computer instruction comprises:

a branch instruction that causes an executing instruction stream to branch to an instruction at an address specified in the instruction if a state of a specified state name is a specified value.

2. The instruction of claim 1 wherein the state is set to a logic one or a logic zero by a processor and indicates the currently processing state.

3. The instruction of claim 1 wherein the state is set to logic one or a logic zero by a microengine in a parallel processor and indicates the currently processing state.

4. The instruction of claim 2 wherein the state is available to all microengines.

5. The instruction of claim 1 further comprising:

an optional token that is set by a programmer and specifies a number i of instructions to execute following the branch instruction before performing the branch operation where the number of instructions can be specified as one, two or three.

6. The instruction of claim 1 wherein the instruction has the following format:

br_inp_state[state_name, label#], optional_token.

7. The instruction of claim 1 wherein the instruction causes a branch if the value of the specified state name is set to a logic one.

8. The instruction of claim 1 wherein the instruction causes a branch if the value of the specified state name is cleared to a logic zero.

9. The instruction of claim 1 wherein the state name is the name assigned to an executing context.

10. A method of operating a processor comprises:

evaluating a value of a specified state name; and

performing a branching operation based on the value of the specified state name being set or cleared.

11. The method of claim 10 wherein the state is set to a logic one or a logic zero by a processor and indicates the currently processing state.

12. The method of claim 11 wherein the state is set to logic one or a logic zero by a microengine in a parallel processor and indicates the currently processing state.

13. The method of claim 11 further comprising:

branching to an instruction at a branch target field specified as a label in the instruction.

14. The method of claim 11 further comprising:

executing a number i of instructions following execution of the branch instruction before performing the branch operation based on evaluating an optional token that is set by a programmer.

15. The method of claim 11 further comprising:

evaluating an optional token that is set by a programmer; and

executing one instruction following execution of the branch instruction before performing the branch operation based on whether the optional token is set.

16. A processor comprises:

a register stack;

an arithmetic logic unit coupled to the register stack and a program control store that stores a branch instruction that causes the processor to:

evaluate a value of a specified state name; and

perform a branching operation based on the value of the specified state name being set or cleared.

17. The processor of claim 16 wherein the processor is a parallel processor with the register stack an arithmetic logic unit is part of a microengine, and the processor further comprises:

at least an additional microengine, the microengine comprising:

a register stack;

an arithmetic logic unit coupled to the register stack and a program control store, and wherein the state is set to logic one or a logic zero by one of the microengines and indicates the currently processing state.

18. The processor of claim 16 further comprising:

a branch target field specified as a label in the instruction.

19. A computer program product residing on a computer readable medium for causing a processor to perform a function comprises instructions causing the processor to:

evaluate a value of a specified state name; and

perform a branching operation based on the value of the specified state name being set or cleared.

20. The computer program product of claim 19 wherein instructions to perform a branching operation branch if the value of the specified state name is set to a logic one.

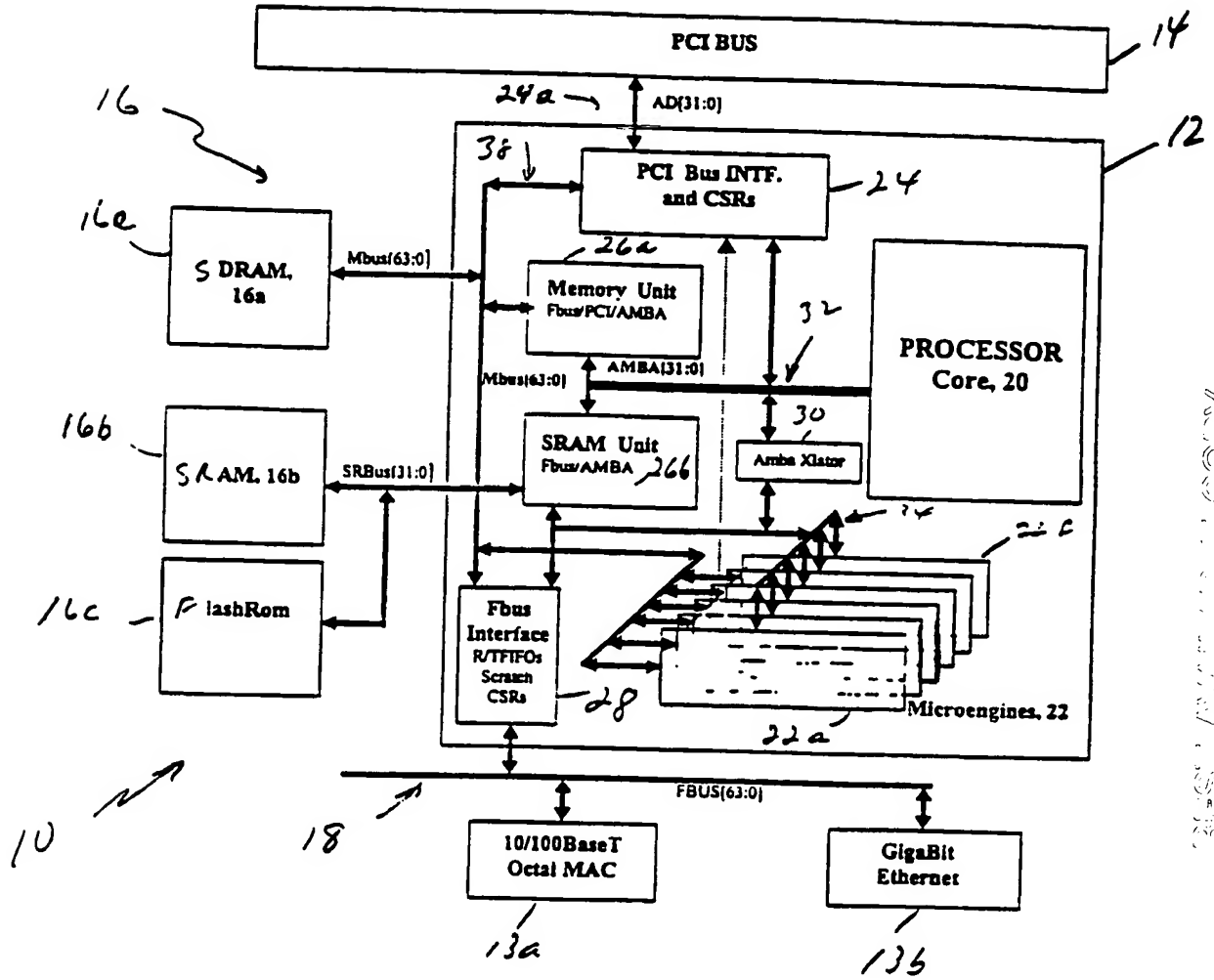

PCI BUS
14
16
24a
AD(31:0)
12
38
PCI Bus INTF. and CSRs
24
16a
SDRAM, 16a
Mbus(63:0)
26a
Memory Unit Fbus/PCI/AMBA
32
PROCESSOR Core, 20
Mbus(63:0)
AMBA(31:0)
16b
SRAM, 16b
SRBus(31:0)
SRAM Unit Fbus/AMBA
26b
30
Amba Xlator
16c
FlashRom
Fbus Interface R/TFIFOs Scratch CSRs
28
Microengines, 22
22a
10
18
FBUS(63:0)
10/100BaseT Octal MAC
GigaBit Ethernet
13a
13b

FIG. 1

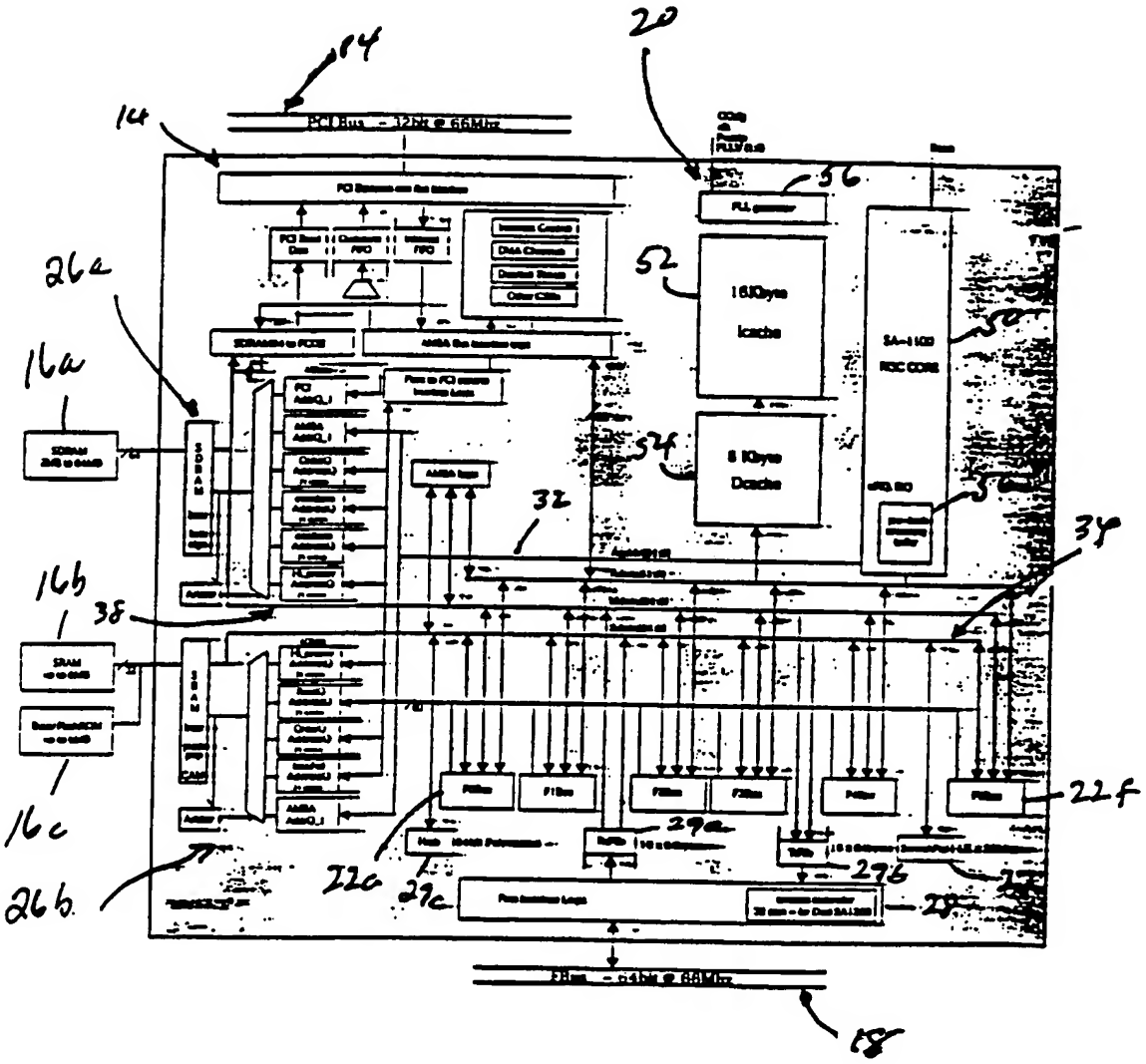

14
20
24
26a
16a
16b
16c
26b
52
54
56
32
34
38
22a
22f
29a
29c
27b
18
16kbyte
Icache
8 Kbyte
Dcache

FIG. 2

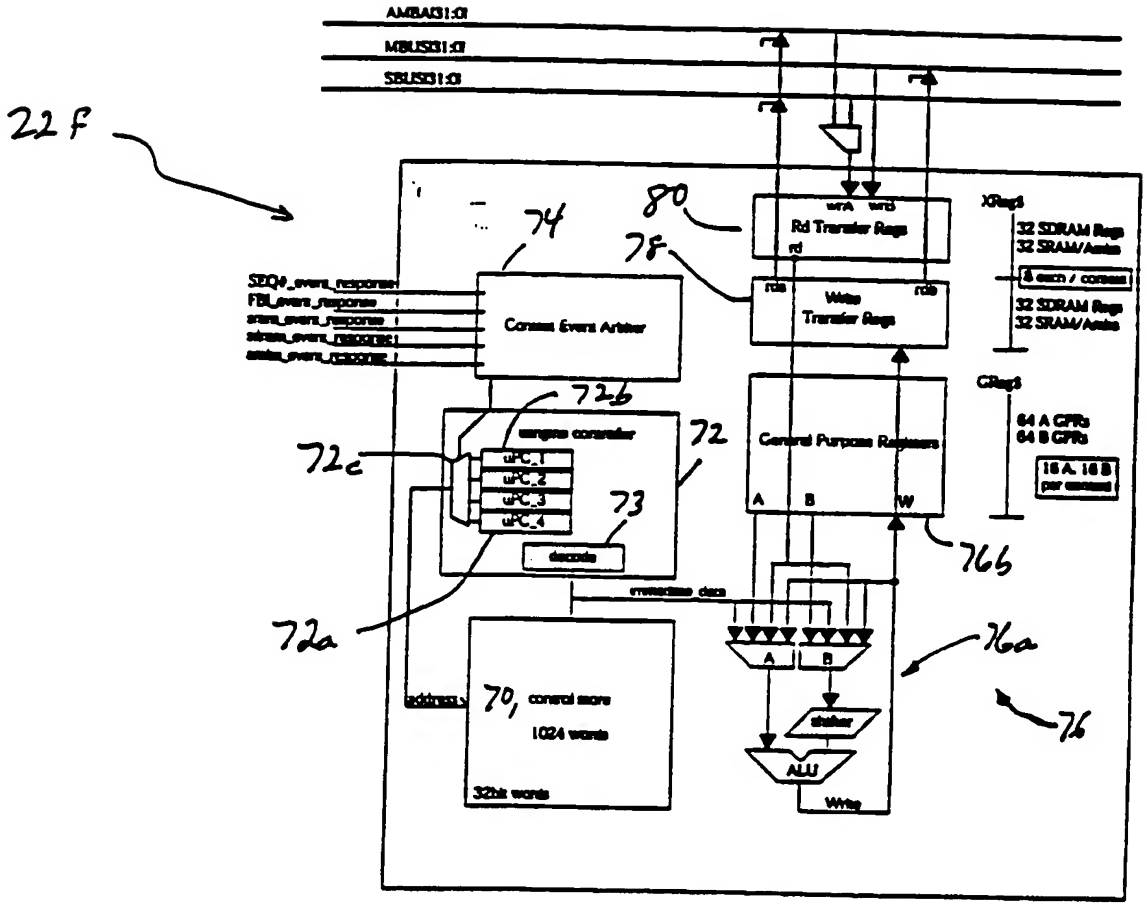

22f
74
80
78
72b
72
72c
73
72a
76b
76a
76
70
Rd Transfer Regs
Write Transfer Regs
Consec Events Arbiter
General Purpose Registers
A
B
W
ALU
1024 words
32bit words

FIG. 3

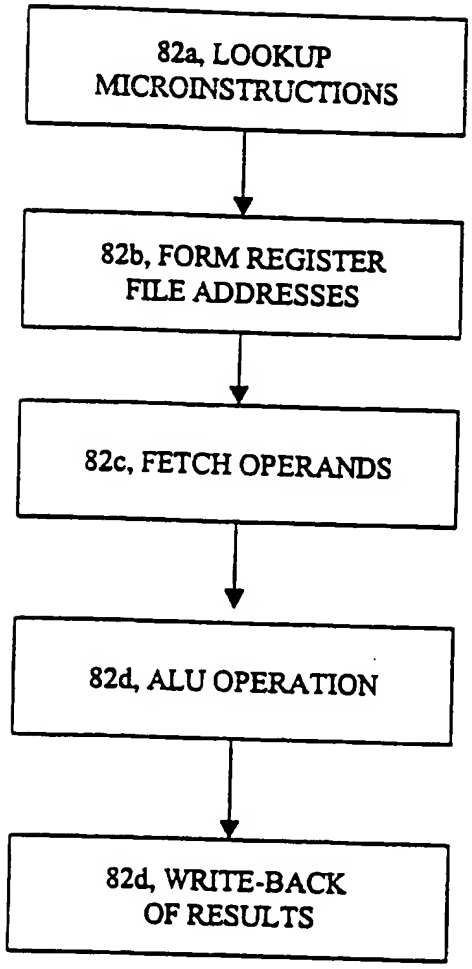
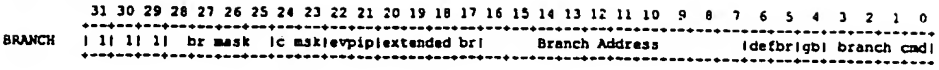

82a, LOOKUP MICROINSTRUCTIONS
82b, FORM REGISTER FILE ADDRESSES
82c, FETCH OPERANDS
82d, ALU OPERATION
82d, WRITE-BACK OF RESULTS

FIG. 4

branch instruction:



defbr: A value of 0, 1 or 2 may be specified. If non-zero, the value indicates that the following 1 or 2 microwords will be allowed to execute before the branch operation takes place.

gb: If set, guess that the branch path will be taken, thus prefetch the branch microword address. Otherwise prefetch the non-branch path. This field is only allowed to be set when defbr=0 or defbr=1.

branch address: branch address conditionally or unconditionally selected.

br_mask: Is decoded to the following options:
1) unconditional branch
2) branch when ALU<31>=1 (<0)
3) branch when ALU<31>=0 (>=0)
4) branch when ALU<31>=1 OR ALU<31:0>=0 (<=0)
5) branch when ALU<31>=0 AND ALU<31:0>!=0 (>0)
6) branch when ALU<31:0>=0 (=0)
7) branch when ALU<31:0>=1 (!=0)
8) branch when specified context mask = current context
9) branch when specified context mask != current context
10) branch on carry-out set
11) branch on carry-out clear
15) look at extended branch field to further decode branch type

extended_br: branches on various context-swapping signals or other signals.

evpip: indicates pipe stage that this branch should be evaluated in

c msk: specifies a context number with which to conditionally branch on.

branch cmd further specifies the type of branch, e.g., looks at condition codes or some other branch criteria.

FIG. 5

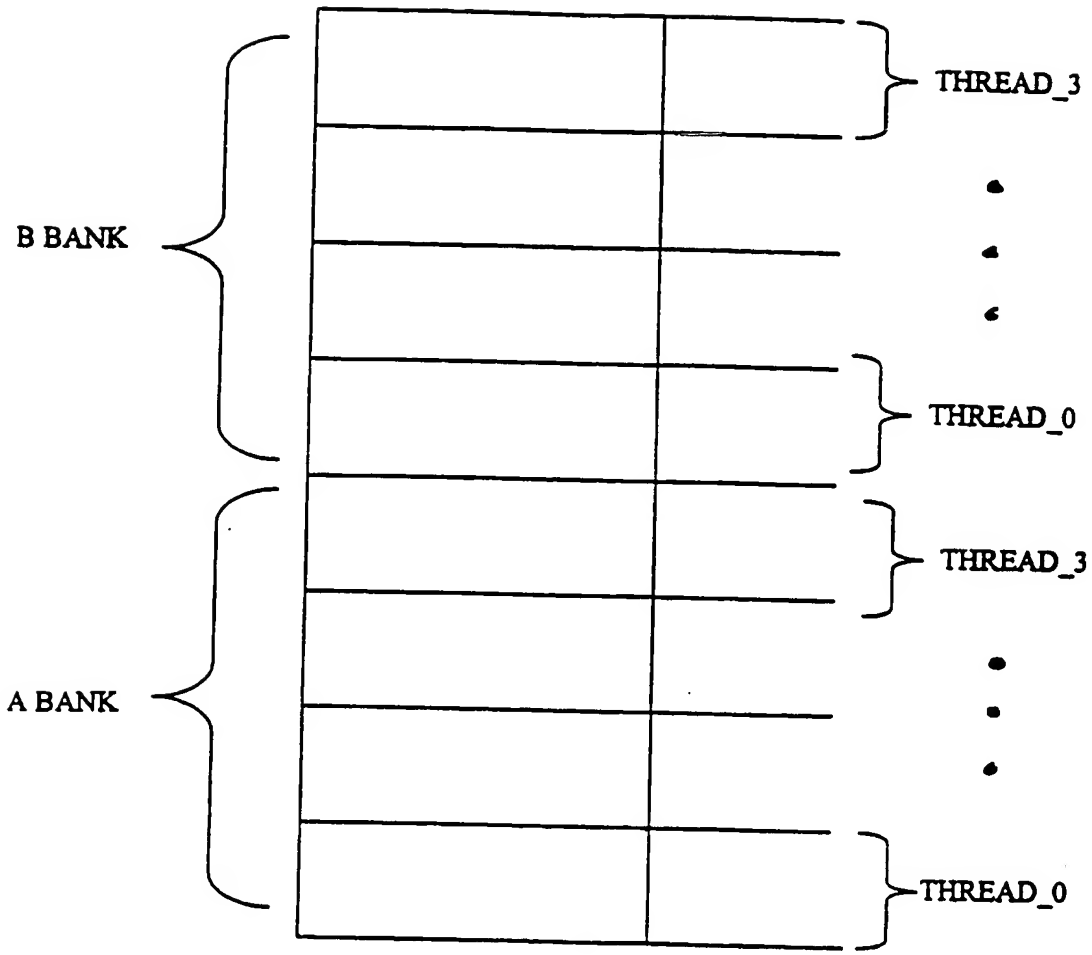


FIG. 6

INTERNATIONAL SEARCH REPORT

International application No.
PCT/US00/23983

A. CLASSIFICATION OF SUBJECT MATTER
IPC(7) :G06F 9/32, 9/26
US CL : 712/233, 234
According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
U.S. : 712/233, 234

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X ---- Y	US 5,517,628 A (MORRISON et al) 14 May 1996, col. 50, lines 10-24; col. 49, lines 35-46; Abstract; col. 1, lines 19-25; col. 33, lines 59-67; col. 44, line 61 - col. 45, line 12; col. 49, lines 35-46; col. 50, lines 10-67; col. 52, lines 4-8; Figs. 6-7 and 19.	1-2, 7-11, 16 and 19-20 ----- 1-20
Y	US 5,659,722 A (BLANER et al) 19 August 1997, Abstract; Tables 1 and 2; Fig. 6	1-20
Y	US 5,428,809 A [Coffin et al] 27 June 1995, Abstract; col. 3, lines 20-32	3-4, 12 and 16-17

☐ Further documents are listed in the continuation of Box C. ☐ See patent family annex.

*	Special categories of cited documents:	"T"	later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention
"A"	document defining the general state of the art which is not considered to be of particular relevance	"X"	document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone
"E"	earlier document published on or after the international filing date	"Y"	document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art
"L"	document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)	"&"	document member of the same patent family
"O"	document referring to an oral disclosure, use, exhibition or other means		
"P"	document published prior to the international filing date but later than the priority date claimed		

Date of the actual completion of the international search 05 OCTOBER 2000	Date of mailing of the international search report 27 NOV 2000
Name and mailing address of the ISA/US Commissioner of Patents and Trademarks Box PCT Washington, D.C. 20231 Facsimile No. (703) 306-5404	Authorized officer WEN-TAI LIN Telephone No. (703) 305-4875

Form PCT/ISA/210 (second sheet) (July 1998)★